TaskInsight: Understanding Task Schedules Effects on Memory and Performance

Germán Ceballos†, Thomas Grass‡, Andra Hugo† and David Black-Schaffer†
† Dept. of Information Technology, Uppsala University, Sweden
‡ Barcelona Supercomputing Center, Spain
german.ceballos@it.uu.se, thomas.grass@bsc.es, andra.hugo@it.uu.se, david.black-schaffer@it.uu.se

ABSTRACT

Recent scheduling heuristics for task-based applications have managed to improve their by taking into account memory-related properties such as data locality and cache sharing. However, there is still a general lack of tools that can provide insights into why, and where, different schedulers improve memory behavior, and how this is related to the applications' performance.

To address this, we present TaskInsight, a technique to characterize the memory behavior of different task schedulers through the analysis of data reuse between tasks. TaskInsight provides high-level, quantitative information that can be correlated with tasks' performance variation over time to understand data reuse through the caches due to scheduling choices. TaskInsight is useful to diagnose and identify *which* scheduling decisions affected performance, *when* were they taken, and *why* the performance changed, both in single and multi-threaded executions.

We demonstrate how TaskInsight can diagnose examples where poor scheduling caused over 10% difference in performance for tasks of the same type, due to changes in the tasks' data reuse through the private and shared caches, in single and multi-threaded executions of the same application. This flexible insight is key for optimization in many contexts, including data locality, throughput, memory footprint or even energy efficiency.

Keywords

Task-based Scheduling; Data Reuse; Data Locality

1. INTRODUCTION

Scheduling task-based applications has become significantly more difficult due to the growing complexity of computer architectures. Typical approaches for optimizing scheduling algorithms consist of either providing an interactive visualization of the execution trace [5, 1] or simulating the tasks execution to evaluate the overall scheduling policy in a controlled environment [12, 4]. The developer then has to analyze the resulting profiling information and deduce if the scheduler behaves as expected, and *qualitatively* compare different schedulers.

PMAM'17, February 04-08 2017, Austin, TX, USA

ISBN 978-1-4503-4883-6/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3026937.3026943

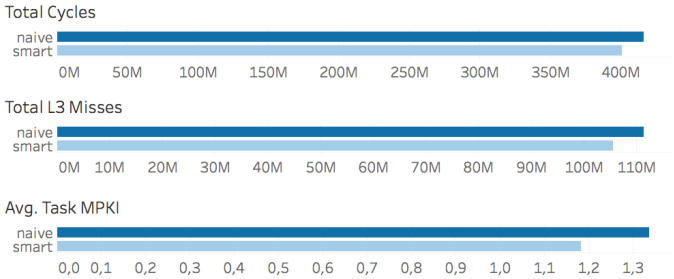


Figure 1: Performance difference between `smart` and `naive`.

Poor scheduling decisions often cause performance variations across tasks of the same type, which makes it hard to identify the root cause from the overall schedule. Existent work [13] proposed scheduling strategies that include these performance differences in the load-balancing algorithm to overcome this problem. However, understanding the underlying causes of performance anomalies of the tasks as well as the snowball effect of the dynamic scheduler is still an open question.

The effects of poor scheduling decisions can be most easily seen in idle execution time due to load imbalance from the inability to prioritize tasks on the critical path or appropriately map tasks to processors. However, scheduler decisions also impact data locality in the cache hierarchy by changing the order of producer and consumer tasks. The result of these decisions is performance variation across tasks of the same type, which can only be understood by analyzing how the tasks share data and how the schedule affects that sharing.

Generally, task-based application developers blame this performance degradation on data locality and attempt to characterize their workload based on data reuse without considering the dynamic interaction between the scheduler and the caches [14, 3]. This is simply because there has been no way to obtain precise information on how the data was reused through the execution of an application, such as how long it remained in the caches, and how the scheduling decisions influenced the reuse history. Without an automatic tool capable of providing insight as to whether and where the scheduler misbehaved, the programmer must rely primarily on intuition to understand and adjust the scheduler for improved performance.

In this paper, we present TaskInsight, a new methodology to characterize, in a *quantifiable* way, the scheduling process in the context of one of the most important performance-

related characteristics: how the schedule affects data reuse between tasks through the cache hierarchy. We show how the reuse of data throughout the execution can provide insights into the performance of the scheduler, regardless if it is optimized for data locality, bandwidth, memory footprint, etc. Further, TaskInsight can interface directly with the task-based runtime system to provide this information both to the programmer and the scheduler.

Previous work [9] has shown the effects of data reuse distances in performance degradation. Those results were based on aggregated statistics and do not provide the necessary detail to manually (developer) or automatically (runtime system) adjust the schedule to improve performance or locality. Scheduler optimization is a notoriously difficult problem as past decisions affect choices and performance in the future, making it hard to explain performance without a detailed view across the program.

In order to understand the performance of a particular schedule, and thereby the scheduler itself, it is therefore necessary to address three critical questions: (Q1) **What** scheduling decisions influenced the performance of the execution?, (Q2) **When** did those decisions happen? and (Q3) **Why** did those decisions affect the performance?.

Answering these questions is vital for dynamic scheduling strategies that adjust their decisions in real time based on how tasks use the hardware resources. Scheduling decisions need to take into account the individual task performance to optimize the overall application, which is nearly impossible without answers to the above questions.

In this paper we introduce the TaskInsight methodology, which shows how data reuses between tasks can provide key information for answering these questions, as they can be quantified in time, and thereby expose the interactions between the tasks' performance and their schedule. We make the following contributions:

1. A novel classification of the data of each task based on when the data is used over time. This classification is able to expose different memory behaviors inherent to the schedule.
2. A new analysis of schedulers based on the preservation of *temporal locality* of the data through time, by connecting our classification to the measured performance results and statistics from the private caches.
3. A new technique to analyze schedulers based on the preservation of *spatial locality* of the data through time, by linking our classification with performance results and statistics from the shared caches.

We start with an example that shows how the overall performance of an application changes when executing with different schedules due to an increase in last-level cache misses (Section 2). We then propose a profiling tool and TaskInsight's data classification technique that allows to clearly differentiate the schedules in terms of their data reuse patterns, using a data reuse graph as in [2] (Section 3). Later, we show how to connect this classification to changes in data reuse, changes in cache misses and changes in performance during the execution: first from the perspective of the private caches (temporal locality on a single-threaded execution, Section 4) and later from the shared caches (spatial locality on multi-threaded run, Section 5).

2. MOTIVATION

It is well known that cache optimization is crucial for performance, and we begin by illustrating these effects on a task-based setting to understand the main driver behind TaskInsight.

We consider a simple example in a simulated environment that allows us to precisely control the effects of memory bottlenecks: a task-based implementation of the Cholesky Factorization using the OmpSs runtime [6]. The input is a 32MB matrix with 256x256 block size, which is enough to hold one task's dataset at a time in 2MB last level cache. The application generates a total of 120 tasks of four different types (`gemm`, `potrf`, `syrk`, and `trsm`). We study the performance over time in a simulated[1] single-threaded execution using the TaskSim simulator [11, 10].

Figure 1[2] shows the total cycle count (execution time of the tasks), total number of last level cache misses and average task last level cache misses-per-kilo-instruction (`mpki`), for two different executed schedules, provided by the OmpSs runtime. The first policy, `naive`, uses a breadth-first-search policy, scheduling tasks in creation order, while `smart` schedules tasks according to a heuristic, wherein child tasks are prioritized over the next task in the breadth-first order. This heuristic optimizes for locality, as child tasks are more likely to reuse data from their parent.

Note that we are reporting performance in terms of *task cycle count*. This metric counts just the time it takes to execute the tasks, and ignores overheads for the scheduler, runtime system, and load imbalance. By measuring just the task cycle count, we can accurately capture changes in the performance of specific tasks, which is essential to understand the impact of the scheduler on task performance.

As we see from the total cycle count, `smart` is 6% faster than the `naive` scheduler. From the cycle count breakdown we see that the main difference comes from the number of cycles spent on DRAM accesses as a result of a 5% increase in last-level cache misses. This results in a 14% increase in the average task MPKI. However, while these statistics clearly show that the memory behavior and performance are affected by the scheduling choice, the overall statistics are not detailed enough to provide actionable insight into the reasons for the increase in cache misses, when they happen, or what potential there is for improvement.

Existing tools follow a similar approach and propose analyzing the performance of the overall application [4] or of the different tasks independently [5]. However, they lack information on how the tasks influence one another through the execution (schedule and caches) and how the algorithmic decisions of the scheduler can impact the performance of the tasks and thus of the application. Such information is necessary not only for evaluating performance, but also for improving the scheduler.

TaskInsight covers this missing area by inferring quantitative metrics about the scheduler's impact on data reuse between tasks and connecting it to performance analysis of different schedules.

In the next section, we show how we can combine the execution schedule with a new data classification to build a data reuse graph, which exposes both *potential* and *actual* data reuse in a hardware-independent manner. We then analyze the behavior of the schedule under specific hardware configurations, such as cache sizes, to determine the schedule's impact on a given system. This flexibility allows us to

[1]For the case study, we chose the default configuration, using a 2MB last level shared cache. Section 5 shows results for native runs on real hardware.

[2]The reader can click on the figures to see an online interactive version of the data.

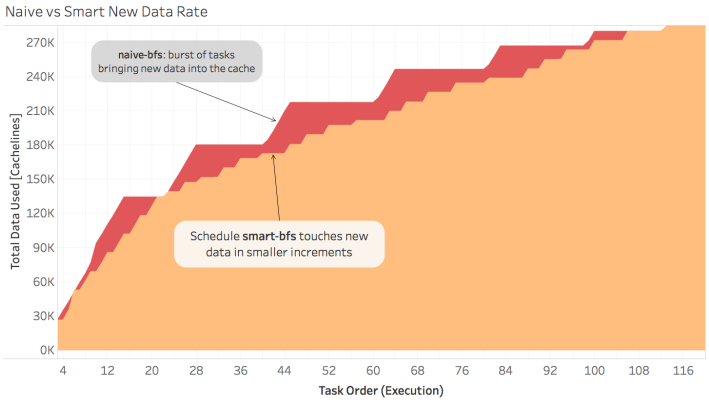


Figure 2: Differences in new data rate.

explain not only the impact of different schedules on different systems, but to also work backwards to understand how earlier scheduling decisions affected the performance of later tasks due to data reuse through the caches.

3. THROUGH THE DATA-REUSE GLASS

Typically each task in an application operates on its own data set, but, over time, parts of a task's data may be reused by later tasks. This means that a portion of the data set can be considered *shared*. If the scheduler can arrange to execute the tasks close enough together in time, it will increase the chance that the shared data is in the cache, and thereby improve performance through temporal locality.

To understand the impact of these scheduling decisions, it is first necessary to analyze how much *shared* and *private* data each task has. TaskInsight does this by profiling the execution to sample the memory addresses for each task. Once the profile is collected, TaskInsight makes the following classification: For a given schedule, every memory access for a task is either new (first time seen) or reused from a previous task. With this observation, we can divide memory accesses into the following four categories:

- `new-data`: the first time the memory address is used in the application.
- `last-reuse`: the memory address was used by the immediately previous task before the current one.
- `2nd-last-reuse`: the memory address was used by the second-to-last task before the current one.
- `older-reuse`: the address was used by a task that came more than two tasks previous to the current one.

Figure 2 compares the cumulative amount of data touched as the program executes between the two schedules from Figure 1. Note that the total number of accesses in the `new-data` category is a function of how much data the application uses, and, as a result, both schedules bring in the same total by the end of execution. In Figure 2 we can see how the `naive` schedule (red curve) executes tasks in a way that touches new data much more aggressively, in bursts. On the other hand, the `smart` schedule (orange curve) is much smoother, meaning that new data is brought at a slower rate. The flat regions on the curves indicate *reuse-periods*, where the scheduled tasks operate on previously used data, and therefore do not bring in any `new-data`. From this data it is clear that the different schedules result in tasks reusing data in significantly different patterns, which will clearly result in different cache miss rates, and therefore impact performance when executed.

Although the `new-data` category intuitively exposes the *rate* at which the applications install new data in the caches, it does not explain how the shared data is used. Thus, to understand the details of how the two schedules reuse data differently, we need to look at the other memory access categories.

Figures 3a and 3b show the breakdown of memory accesses (`new-data`, `last-reuse`, `2nd-last-reuse`, `older-reuse`) for both schedules (`naive`, `smart`) as a function of time (task scheduled). The `last-reuses` are shown on the bottom (dark blue), while the upper area (orange) represents the new data regions. The lower-middle region shows the percentage of second-last memory accesses, and finally the upper-middle region (light blue) displays the relative amount of data reuses that come from older tasks, `older-reuse`.

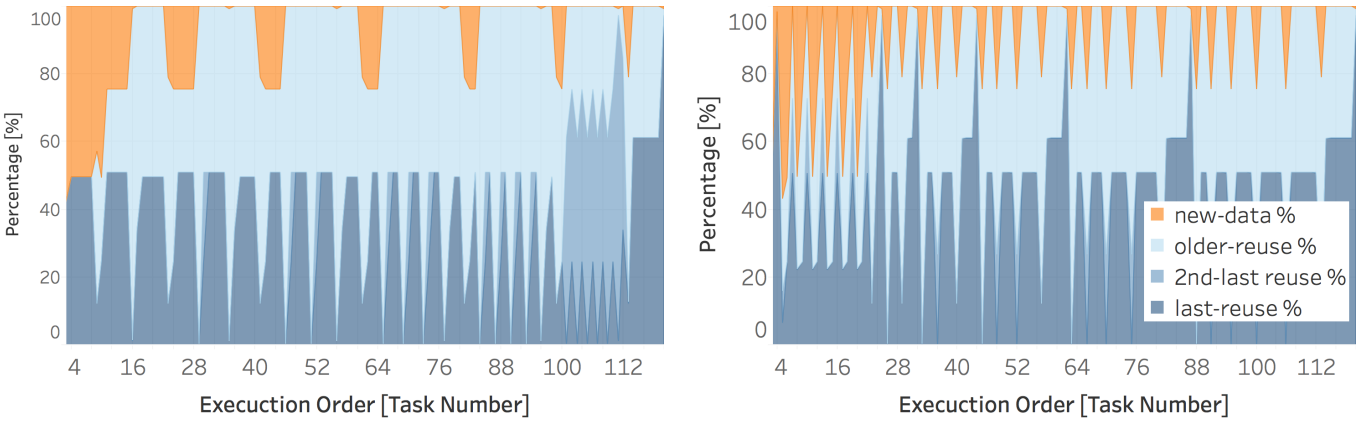
The first thing we notice is that the area corresponding to `new-data` is distributed more sparsely across the graph for the `smart` policy, compared to the `naive` approach, which touches most `new-data` during the first 16 tasks. In addition, the area corresponding to last-reuses increases considerably (more dark blue area) in the `smart` schedule, meaning that more shared data is being reused sooner. This is also observed between tasks 100 to 115: in the `naive` schedule, most of the data used is coming from the second-last predecessor, but in the `smart` schedule, data is coming from its immediate predecessor. As the immediate predecessor is more recent, one would intuitively expect that this schedule would result in a higher cache hit rate and better performance.

With this classification, it is now clear that the two schedules have very different reuse characteristics. However, we need to translate the observations from the previous figures into relevant metrics to compare the schedules overall. TaskInsight uses aggregated statistics from each reuse category over time to understand how reuses flow from one category to another.

Figure 4 shows an example of this. It displays the percentage of memory accesses corresponding to each category (y-axis, %), as a function of time (x-axis; task number), for both the `naive` (left) `smart` (right) schedules. The average value (%) for each access category is displayed with the *Average* line. By comparing the averages, it is possible to see that the `smart` scheduler has 11% more `last-reuses` than `naive`. Most importantly, this view of the execution allows us to understand the effect of these changes: we can see that 5% of the execution time increase comes from the `smart` schedule turning `2nd-last-reuses` into `last-reuses`, while the remaining 6% comes from improving older reuses.

Figure 4 not only allows us to see *what* aspects of the schedules are different, but also to precisely detect *when* the schedules have differences in data reuse. Since the tasks are uniquely identified, it is possible to point to the specific tasks that benefited from the rescheduling or were hurt by it. In the following section, we show how to connect this classification with performance measurements to also understand *why* performance was affected by changes in memory behavior due to scheduling.

Overall, the TaskInsight analysis allows us to understand the impact of scheduling changes in a way that can be used to improve performance by increasing reuse through caches. Approaches that only measure the actual cache miss ratios per task (e.g., hardware performance counters) are unable to trace back changes in memory behavior to the scheduling decision that caused them in this manner. As a result,



(a) Data reuse for `naive` schedule: new data is brought in big chunks, and data is reused far away, exposing more `older-reuses` during the execution.

(b) Data reuse for `smart` schedule: new data is brought more sparsely in smaller chunks, and data is reused sooner showing more `last-reuses`.

Figure 3: Relative data reuse.

this novel methodology enables scheduler designers to gain insight into how specific scheduling decisions impact later tasks.

4. ANALYZING PERFORMANCE

The classification described in the previous section allows us to characterize the impact of different schedules on memory behavior in a hardware-agnostic manner. However, comparing the relative differences between these metrics is not enough to predict how they will affect performance. To accomplish this TaskInsight combines the data reuse classification with performance measurements to explain changes in performance due to changes in data access.

We will first use TaskInsight to study how scheduling affects performance due to changes in the data's temporal locality, and how it is related to cache reuse. To illustrate the analysis, we consider the simulated single-threaded execution of the Cholesky factorization from Section 2, which exposed a performance difference between two schedules. Focusing on single threaded execution allows us to exclude contention for the shared last-level cache. Section 5 later extends TaskInsight to analyze multi-threaded runs, both regarding temporal and spatial locality changes due to scheduling, and explaining performance changes.

During the Cholesky factorization execution, we measure the performance of each task (average cycles-per-instruction, CPI) and cache behavior such as last-level cache misses and accesses. The CPI for each task instance (color-coded by type) is shown in the bottom of Figure 4. We can see that the performance across tasks (CPIs) vary far more in the `naive` schedule (average 20% worse). For instance, if we look at Task with ID 57, in `naive` it was scheduled as the 57th task, executing at 0.33 CPI. In `smart`, this task was executed 32nd, delivering a 15% increase in performance at 0.29 CPI.

As TaskInsight can tell exactly where the data is coming from, by correlating CPI with the data reuse classification, it is possible to see that in the first schedule the task is reusing 98.1% of its data from older tasks (highlighted in the figure). Scheduling this task sooner results in 96.5% of the data is coming from the previously executed task, increasing the likelihood that the data is reused through the cache, and thereby increasing performance.

Moreover, we also see variations in task performance within the same schedule for tasks of the same type: e.g. in `smart`, tasks 113-118 are all `syrk` with same input size, however, task 113 has a 7% worse CPI than its subsequent ones. Again, by correlating with the `new-data` graph, we see that this is because it is bringing 20% new data for that schedule.

In addition, by knowing the size of the last level cache, and by looking at the amount of new data per task, it is possible to estimate how many tasks can be scheduled, and which of them, before the data that is going to be reused is evicted. When optimizing for locality, this kind of insight is critical: if data is never going to be touched again, it is possible to schedule a task that brings new data into the cache; if the data is not going to fit in the cache anyways, tasks can be scheduled later in order to prioritize those that reuse data already present. This information can also be used to determine the task granularity (size), as the amount of reuse that can be realized through the cache depends on the size of the tasks' data.

5. MULTI-THREADED EXECUTIONS

The previous section showed how TaskInsight can quantitatively characterize different schedules with regards to their *temporal data locality* and data reuse through the caches for single-threaded applications. However, and unlike previous methods such as looking at aggregated hardware performance counters, our technique enables us to identify *which* specific scheduling decisions caused changes in memory behavior, *when* they occurred, and *why* they lead to performance loss.

When running multi-threaded applications, the complexity of the analysis is significantly increased by having multiple *per core* schedules executing in parallel and the effects of shared caches, which can cause the schedules to interfere with each other. The shared cache effects can be particularly important for performance as the cost of an L3 miss is much higher than an L2 miss (and L3 hit).

When using TaskInsight for multi-threaded executions, it provides information about the changes in temporal locality on a per core basis. However, to handle parallel executions of multiple schedules it is also necessary to understand the effects in performance of the shared cache.

As multi-threaded executions can execute tasks at a far higher rate, it is often difficult to identify tasks whose performance was affected by the schedule. We begin by showing how to use *performance-to-memory correlation* as a filtering technique (Section 5.1). We then explain how to use Task-

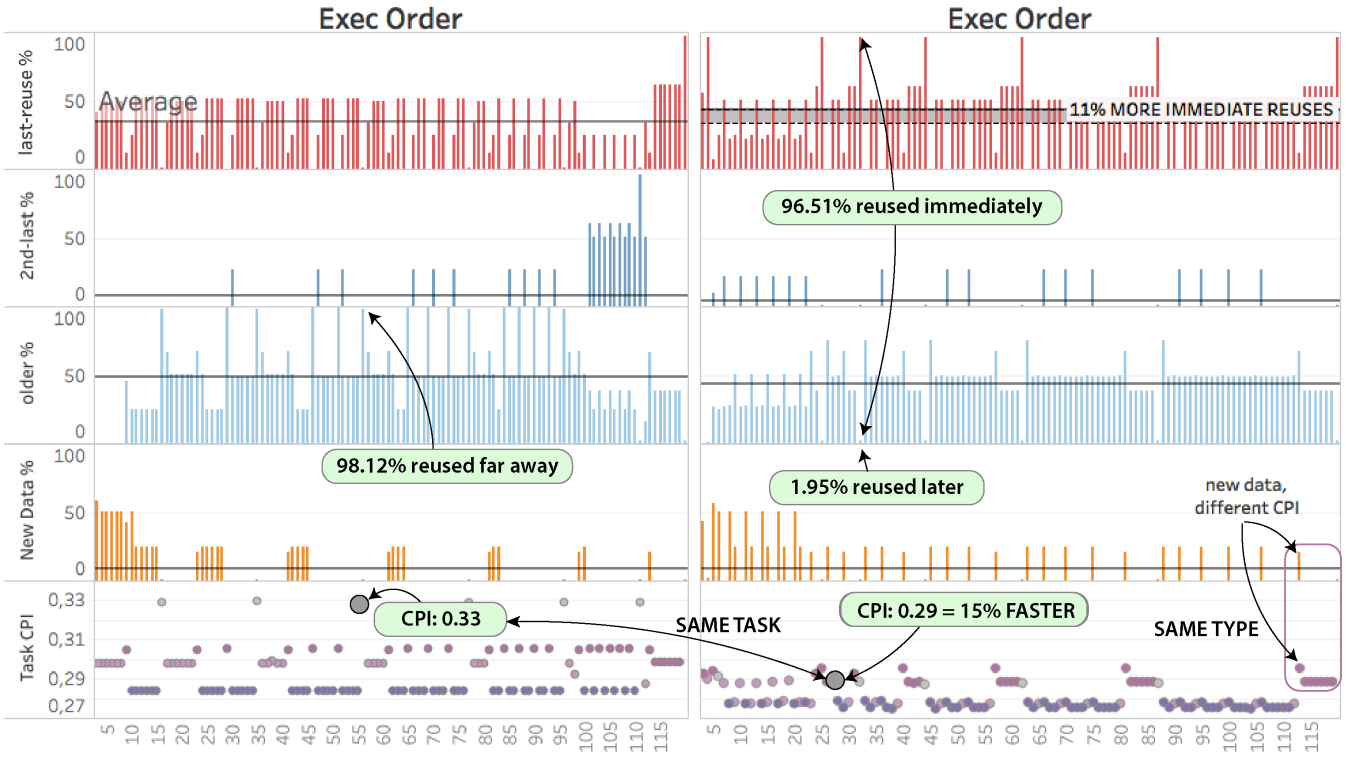


Figure 4: Breakdown of reuses per category: `naive` (left) vs `smart` (right).

Insight on a per-core basis to reveal where schedules fail to preserve temporal data locality through the private caches (Section 5.2). Finally, we show how to extend TaskInsight to explain the effects of shared caches in performance variation, by modeling also spatial data locality of each schedule (Section 5.3).

5.1 Performance-memory filtering

For our analysis of parallel tasks we now consider a larger execution of the same Cholesky factorization on a quad-core machine[3]. The execution consists of 796 task instances in total, working with a 256x256 block size, where 550 are `dgemm` (80% of the total cycle count), 114 are `dsyrk`, 117 are `dtrsm` and 15 are `dpotrf`. Each of these task routines are from the Intel MKL library.

We executed the application with the same two scheduling policies as before (`naive` and `smart`) on 4 cores[4], and collected data from hardware performance counters including number of instructions, cycles, and L2/L3 cache misses/accesses, *per-task* (implementation details in Section 6).

When aggregating these results, we see that the total cycles speedup of `smart` against `naive` is over 10%, and when looking at the cycle breakdown, `naive` incurs 9% more L2 misses on average per core, and 40% more L3 misses than `smart`. This is a significant variation due to scheduling.

TaskInsight can determine which specific tasks were affected by the schedule differences and at which specific moments in time. However, displaying data from a large multi-threaded execution would require very complex graphs for each execution core. To see the effect of scheduling and task type on memory related performance, we start by *correlating performance to memory* on a per task basis in Figures 5a and 5b. This presentation allows to identify which task instances had the most performance variation between the different schedules. These figures show scatter plots of per-task CPIs vs. (private) L2 miss ratios and (shared) L3 miss ratios, respectively, colored by task type for both schedules.

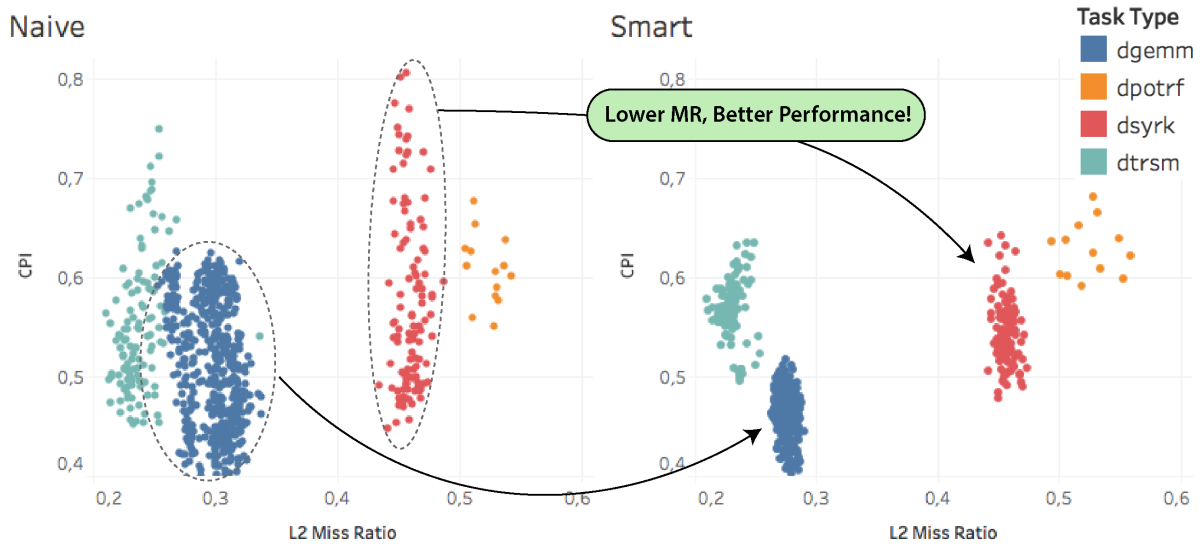
The performance-to-L2 miss ratio data (Figure 5a) shows which tasks are sensitive to temporal locality changes through the L2 (sensitivities to the shared L3 are discussed in Section 5.3). As we can see, the variance in the performance of the tasks using the `naive` schedule (Figure 5a left) is significantly higher, especially for task types such as `dgemm` (blue) and `dsyrk` (red). When we compare this to the `smart` schedule (Figure 5a right), we see that `dgemm` (blue), which represents 80% of the total execution time, have over 11% better CPI as a result of an 8% lower L2 miss ratio. This is seen in how the `dgemm` cluster moves down (lower CPI, better performance) and to the left (lower L2 miss ratio) between the `naive` (left) and `smart` (right) plots in Figure 5a. A similar effect of the `smart` scheduler can be seen for the `dsyrk` (red) task cluster.

The opposite effect can be seen, though, for the `dtrsm` tasks (light-green). In the `smart` case, the cluster shows 4% more L2 misses, causing performance to be 6% worse. Despite this, the overall performance is still better for this schedule as the `dtrsm` tasks represent only 8% of the total execution time. In this case the scheduler has effectively traded off worse temporal locality in the less frequent `dtrsm` for better locality in the far more frequent `dgemm` tasks.

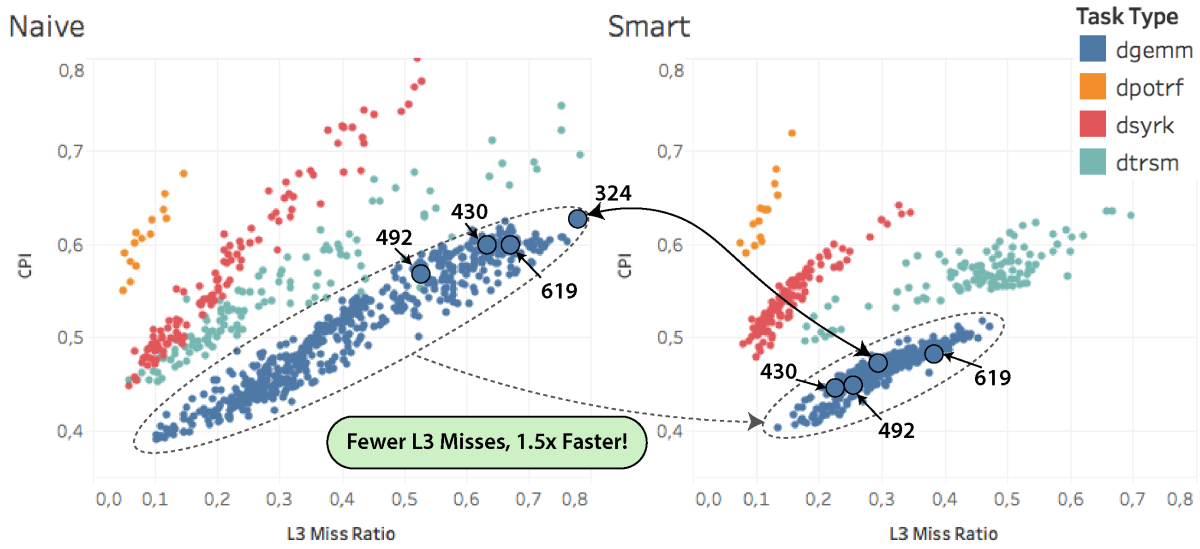
From this correlation we can see both which task types are the most sensitive to memory effects due to scheduling, and also which tasks instances' performance is most likely changed due to L2 caching effects and independent of

[3]Intel Core i5-3550 CPU (Ivy Bridge). 8-way associative L1 cache, 256kB 8-way associative L2 cache, 6MB 12-way associative L3 cache, 16GB RAM.

[4]One thread was pinned per core, and HyperThreading was disabled.



(a) Performance-L2 correlation. Tasks from the same type are clustered better in the `smart` schedule, meaning less performance variation within the same type due to memory.



(b) Performance-L3 correlation. Tasks and types exposing drastic performance degradation due to sensitivity to the shared cache. The `smart` schedule shows tasks with 50% less misses and 50% less performance variation within the same type.

Figure 5: Performance-Memory Correlation.

caching effects. By using this filtering technique to identify these tasks, we can further study them with TaskInsight's technique from Section 4 to fully understand if the changes in performance are due to reuses from L2, as we show in the following section.

5.2 Temporal Locality of Private Caches

As each task is uniquely identified, it is possible to compare the difference of CPI and L2 miss ratio values between the two schedules on a per-task basis. This correlation allows us to distinguish between task instances without performance variation, task instances with performance variation due to non-memory effects (different CPI but no changes in L2/L3 miss ratios) and task instances with performance variation due to memory effects (both different in CPI and miss ratio). By filtering the analyzed tasks to the cases where performance variation is correlated with memory effects, we can target TaskInsight to study only on those tasks where we can benefit from scheduling insight as to how the schedule affected data reuse through the caches.

To analyze private caches, we can use the analysis from Section 4 on a per-core basis, as the private cache behavior is only affected by the local core's schedule [5].

This enables the characterization of the two schedules in terms of how well they preserve *temporal locality* through the private caches. By only looking at the filtered tasks from the previous section, TaskInsight is able to show *why* those particular task instances missed more in the private L2 cache and *when*, which is essential to improve scheduling decisions.

5.3 Locality of Shared Caches

To understand how the shared cache affects the performance of different schedules, we extend TaskInsight to provide information regarding how tasks share the last level

[5]We are ignoring evictions due to the inclusive shared L3 cache as its size is far larger than the private L2s.

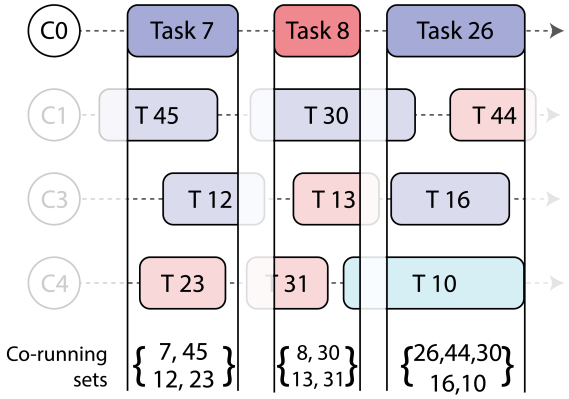


Figure 6: Determining co-running sets: for each core, the set of co-running (overlapping) tasks is identified and used to build a single set with their memory addresses during analysis.

cache, and how this affects performance across different schedulers. The novelty of this approach lies on correlating caching, changes in schedule and changes in performance. This contribution is key to characterize not only how good a schedule is in temporal locality preservation through private caches, but also through shared caches, as well as characterize them in terms of spatial locality.

The correlations between task performance (CPI) and shared cache behavior (L3 Miss Ratio) is shown in Figure 5b, colored by task type. For all task types, there is a rather linear correlation, meaning that the fewer L3 cache misses, the better the performance. Furthermore, we see that almost half of the `dgemm` tasks (blue) experience roughly twice the performance (lower CPI) for the `smart` schedule (right) over the `naive` schedule (left) with L3 miss ratio decreases of up to 50%.

Tasks that have a radical change in L3 misses are likely to have a performance loss, and vice-versa, so we consider those as candidates to study with TaskInsight.

While this performance-based analysis can identify which tasks saw worse performance due increases in L3 misses, it cannot give any insight into *which scheduling decisions* caused them. For example, in the `naive` schedule, two tasks with data reuse between them may have been scheduled further apart from each other, resulting in their data being evicted from the L3 before it could be reused. Alternatively, the tasks may have been executed very close to each other, but co-executed with other tasks that used a large disjoint dataset, thereby polluting the shared cache and evicting the data before it could be shared.

TaskInsight is able to explain where the difference in L3 cache misses is coming from, by modeling each set of co-running tasks, and then computing their predecessor's reuse (temporal reuse) and amount of data reused at the same time (spatial reuse). This now allows us to determine which scheduling decisions cause the increase in miss ratios, which lead to worse performance.

To undertake this analysis, we first look at the sequence of tasks executed on each core, i.e. the per-core schedules. For each task instance, we determine the set of *overlapping tasks* for the schedule, which is the set of tasks executing on other cores at the same time. Figure 6 shows an example of this. In this case, the first co-running set comprises tasks 7, 45, 12 and 23, which are executed in parallel from core 0's perspective. The second set contains tasks 8, 30, 13 and 31, and so on. This analysis gives us a sequence of co-running tasks over time, across all different cores. Each core will have a different sequence, but our analysis generates similar results for each of them.

At the same time, TaskInsight builds the application's (schedule-independent) *reuse graph* introduced in [2], and combines it with the co-running task sequence to compute the set of memory addresses used by each co-running set. This allows to model the sequence of co-running tasks over time, and use the analysis in Section 4 to analyze how much data was reused over time, but in the shared cache.

The result of this analysis is shown in Figure 7: the top half of the graph corresponds to the `naive` schedule and the bottom half to the `smart` schedule. The x-axis represents execution order of the 216 *sets* of co-running tasks. The top of each graph shows the classification of the reuses provided by TaskInsight as in Figure 3.

Similarly to the single-threaded case, there is a noticeable difference in how the two schedules touch new data: `naive` is touching the data used by all the tasks in the first 38 co-running sets (152 tasks), while `smart` is bringing new data into the caches throughout the entire execution. In addition, the `smart` schedule has 11% more `shared-last` reuses (dark blue area) than the `naive` schedule (50% vs 39%), meaning that the co-running tasks are using 11% more data from the previously executed tasks, thereby increasing the likelihood of hitting in the shared cache.

The figure also displays the results of the hardware performance counters in the bottom. The filled area (green) represents the absolute L3 miss ratio for each of the co-running set of tasks, while the line (green) shows the averaged performance (CPI) across all cores per co-running set of tasks. With both the TaskInsight data classification and the hardware performance counter results, we can now see the correlation between the classification of where a schedule's tasks are reusing data from and the performance impact due to the resulting cache behavior. As all the data does not fit entirely in the shared cache, when the `naive` schedule touches all the data upfront, it results in a significant increase in the number of cache misses, and a corresponding decrease in performance for the co-executed tasks (x-axis 43 to 93).

It is also possible to see how an increase in `shared-last` reuses results in a lower L3 miss ratio, and vice-versa, where an increase in `older-reuses` generally results in more L3 misses. Two examples of this are highlighted in the figure, where closer reuses result in fewer cache misses, and farther reuses miss in the cache due to eviction. The `smart` schedule achieves a 20% better L3 miss ratio than the `naive` schedule, yielding a 10% performance improvement. It is only now with TaskInsight's analysis that we can see *what* tasks are involved, *when* their data was reused, *why* a schedule impacted performance if it was beyond the cache size limits.

To demonstrate the value of this new analysis, we look at a significant performance change across schedulers and use our analysis to understand what scheduling decisions led to it. The significant change in the L3 miss ratio in Figure 7 (top, A) corresponds to the 86th set of co-running tasks. This set contains tasks 324, 430, 492 and 619, all of which are of type `dgemm`. From the figure we can see that combined they give a miss ratio of 66%. Figure 8 shows the relative change in CPI and cache behavior for these tasks compared to the average of all instances of this task type in the schedule. For these specific co-executed tasks we see a 13% increase in the private L2 misses and an 80% average increase in L3 misses.

When examining where these task instances were executed

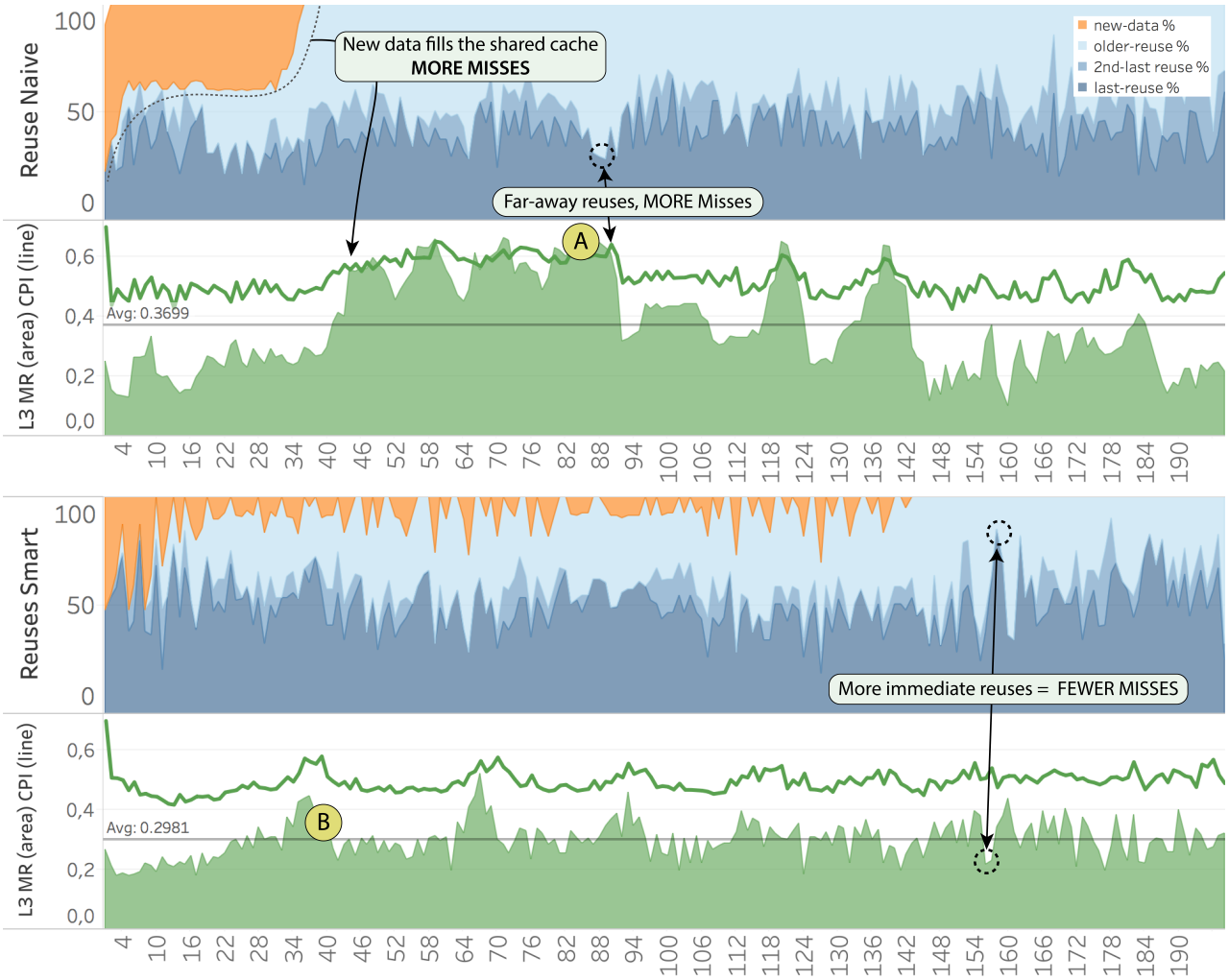


Figure 7: Reuse-Performance correlation. TaskInsight combines data reuse classification with performance and L3 misses to explain scheduling effects at the shared cache.

in the `smart` schedule (Figure 7 bottom, B), we can now see that the source of the problem does not only come from the amount of reuse these tasks have from their previously executed tasks, but also from how they interact at the last level cache. From our analysis, we can compute that the size of the combined datasets of these four task instances is a total of 42K cachelines (roughly 2.7MB). On the other hand, when these tasks were executed in the `smart` schedule, they were overlapped (co-run) with other tasks whose combined datasets were not more than 32K cachelines (2MB). As a result, the smart schedule enabled these tasks to keep their working sets in the L3 cache, while the naive schedule did not. For instance, task 324 was executed in co-running set 22 in `smart`.

This analysis shows that our technique can be used to identify which scheduling decisions result in different memory behavior across tasks. Not only does it give insight into where the data is reused over time, but it can also determine the amount of shared and non-shared data at any point in time, which is vital information to understand if the combined datasets of the co-running tasks fit in the cache, and therefore the effects of scheduling at the shared cache level.

6. IMPLEMENTATION

Figure 9 shows an overview of how TaskInsight combines memory accesses and hardware performance counter information through a profiler, an instrumentation library and an analysis tool. The memory access profiling tool uses Pin [7] to sample the tasks' memory accesses. While profiling, an *address map* is created, between each accessed address (at a cacheline granularity) and all tasks that use it. Note that this also captures execution order (schedule), and therefore provides all necessary information for the classification of the memory accesses.

Next, an instrumentation library collects per-task performance information. The library intercepts runtime calls (OmpSs [6] in our implementation) and records the hardware performance counters at the start and end of each task. The performance counter instrumentation must be run for both schedules, as the hardware results are schedule-dependent, and cannot run at the same time as the memory access profiler due to its performance overhead.

Finally, the analysis tool uses the memory accesses profile, the data from the hardware counters, and the schedule (per core in the multi-threaded case). It builds a *data reuse graph*, connecting datasets and tasks to describe the applications' data characteristics, independent of the schedule. Each node in the data reuse graph represents a task instance, while each edge represents the amount of data sha-

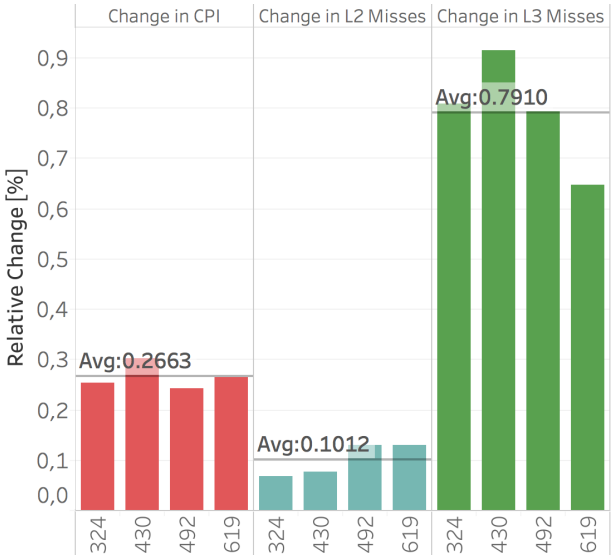


Figure 8: Relative change for overlapping tasks: Four `dgemm` tasks executed together had 80% more L3 misses and 28% worse performance than executing separately under a different schedule.

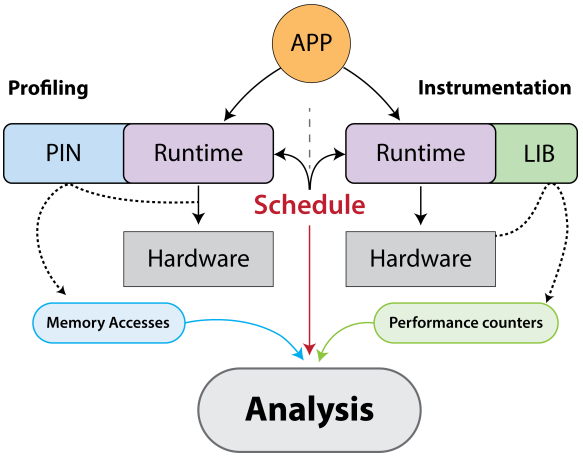


Figure 9: Methodology Overview: TaskInsight consists of a profiling tool, an instrumentation library and an analysis tool.

red between those tasks. In addition, a list of the unique memory addresses (datasets) is kept for each task instance. Combined, these allow us to understand how data is reused for any schedule, and it enables TaskInsight to reconstruct the datasets of any co-running task set.

With the per-core schedule as an input, it is possible to walk through the data reuse graph, analyzing each task's dataset and comparing it to the previous tasks. While the graph is very dense, it is only necessary to walk it according to the input schedule. Furthermore, since the representation is schedule agnostic, it is possible to walk the same graph following a different schedule and generate an analysis for that schedule. This enables the prediction of the sharing under different schedules without the need to re-profile the application.

The TaskInsight methodology outlined here is general, transparent to the applications and independent of the runtime system used, making it directly applicable to any other task-based environment.

7. RELATED WORK

Previous work has proposed different ways to diagnose scheduling anomalies by either interactively visualizing information [5, 1, 8] or by simulating the task execution in order to provide a deterministic behavior of the scheduler [12, 4] without evaluating the performance behavior as a result of how memory is used. Significant work has been done to study the locality as a metric to characterize the workload of an application [14, 3] without considering the scheduling decisions taken as a result of the complex architectures, and only looking at the overall impact of locality on performance [9]. In our work we characterize the scheduling behavior as a result of the memory reuse. We provide quantifiable insight on how two schedulers behave differently through the execution and on how scheduling decisions of a task-based application affect the performance of task instances.

Drebes et al. [5], as well as other visualization tools ([1], [8]) propose summarizing and averaging information provided by both the runtime and the hardware performance counters. By integrating this data in an interactive visualization tool, the programmer can observe the order of execution of the tasks, their duration, data dependencies, status of the computing resources, etc. However, when certain tasks end up executing in a certain order and with different performance, it is up to the programmer to reverse engineer the scheduler's decision, the reasons behind them, and the points where those decisions happened. Our work proposes a solution to help the programmer understand the variation in performance across the tasks, based on the analysis of memory reuse, capable of showing the exact points in time and underlying reasons for this variation.

Stanisic et al. [12] as well as Chronaki et al. [4] rely on simulation of the tasks' execution in order to isolate the scheduler's effect on performance from tasks' unpredictable behavior. In our work we execute the entire application on real hardware and we characterize the interaction between the scheduler and the tasks. Thus we are able to understand how tasks affect each others performance due to memory reuse and how the dynamic scheduling decisions are affected.

Tillenius et al. [13] observed that tasks of the same type have different execution time and estimated task sensitivity to resource sharing. Based on this they adjusted the scheduling in order to optimize the execution time of the application. In our work we analyze the reason for these performance differences, that is the way the tasks are reusing the memory, and we provide information that can be used by the scheduler in order to avoid such effects.

Weinberg et al. [14], as well as Cheveresan et al. [3], propose using *memory reuse* as a metric to characterize workloads. Through this technique they analyze spatial and temporal locality of the application independent of the architecture. In our work we analyze the performance variation of an application when facing dynamic scheduling adaptations on modern architectures. By understanding how the memory is reused through the execution of the application we can evaluate if the scheduler is taking the correct decisions. Unlike [9] we evaluate the reused data throughout the execution and provide an analysis over time. This information can be used by an automatic tool to optimize the performance of the application.

8. CONCLUSION

In this work we presented TaskInsight, a methodology that provides high-level, quantifiable information that ties task scheduling decisions to how tasks reuse data and the resulting task performance. By combining schedule independent memory access profiling (to classify how data is reused between tasks) and schedule specific hardware performance counter data (to determine performance on a given system) we are able to identify *which* scheduling decisions impact performance, *when* they happen, and *why* they cause a problem. TaskInsight goes beyond previous work which typically used aggregate metrics to look at overall memory system behavior or ignored the task-level performance variation due to cache/scheduler interactions. With this deeper insight we can enable future generations of developers and schedulers to better optimize their applications for complex memory systems.

TaskInsight not only gives insight on how a scheduler can be improved but also an explanation for why tasks of the same type can demonstrate significant variation in performance (up to 60% in our examples). As a result, programmers can now quantitatively analyze the behavior of the scheduling algorithm and the runtime can use this information to dynamically make better decisions. TaskInsight diagnoses task scheduling misbehavior for both sequential and parallel applications, and we demonstrated how to use it to understand native multi-threaded executions which expose differences above 10% in performance due to 20% difference in reuses through the private caches and up to 80% difference in reuses through the shared last level cache, caused by scheduling.

By providing this insight into the coupling between the schedule's behavior, data reuse through the cache hierarchy, and the resulting performance, we lay the groundwork for improving scheduling policies.

We are particularly interested in using this information to optimize for locality in NUMA aware architectures, bandwidth in CPU/GPU architectures, memory footprint or even energy efficiency.

9. ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work was supported by the Swedish Research Council, the Swedish Foundation for Strategic Research project FFL12-0051 and carried out within the Linnaeus Centre of Excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center. This paper was also published with the support of the HiPEAC network that received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 687698.

10. REFERENCES

[1] Bell, R., Malony, A. D., and Shende, S. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings* (2003), H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., vol. 2790 of *Lecture Notes in Computer Science*, Springer, pp. 17–26.

[2] Ceballos, G., Hagersten, E., and Black-Schaffer, D. *Formalizing Data Locality in Task Parallel Applications*. Springer International Publishing, Cham, 2016, pp. 43–61.

[3] Cheveresan, R., Ramsay, M., Feucht, C., and Sharapov, I. Characteristics of workloads used in high performance and technical computing. In *Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17-21, 2007* (2007), B. J. Smith, Ed., ACM, pp. 73–82.

[4] Chronaki, K., Rico, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015* (2015), L. N. Bhuyan, F. Chong, and V. Sarkar, Eds., ACM, pp. 329–338.

[5] Drebes, A., Pop, A., Heydemann, K., and Cohen, A. Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016* (2016), IEEE Computer Society, pp. 274–283.

[6] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters 21*, 2 (2011), 173–193.

[7] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), PLDI '05, pp. 190–200.

[8] Müller, M. S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., and Nagel, W. E. Developing scalable applications with vampir, vampirserver and vampirtrace. In *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007* (2007), C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, Eds., vol. 15 of *Advances in Parallel Computing*, IOS Press, pp. 637–644.

[9] Pericàs, M., Amer, A., Taura, K., and Matsuoka, S. Analysis of data reuse in task-parallel runtimes. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation - 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers* (2013), S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds., vol. 8551 of *Lecture Notes in Computer Science*, Springer, pp. 73–87.

[10] Rico, A., Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., Ramírez, A., and Valero, M. On the simulation of large-scale architectures using multiple application abstraction levels. *TACO 8*, 4 (2012), 36.

[11] Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramírez, A., and Valero, M. Trace-driven simulation of multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA* (2011), IEEE Computer Society, pp. 87–96.

[12] Stanisic, L., Thibault, S., Legrand, A., Videau, B., and Méhaut, J. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurrency and Computation: Practice and Experience 27*, 16 (2015), 4075–4090.

[13] Tillenius, M., Larsson, E., Badia, R. M., and Martorell, X. Resource-aware task scheduling. *ACM Trans. Embed. Comput. Syst. 14*, 1 (Jan. 2015), 5:1–5:25.

[14] Weinberg, J., McCracken, M. O., Strohmaier, E., and Snavely, A. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2005), SC '05, IEEE Computer Society, pp. 50–.